Unit-IV: Inheritance

Definition:

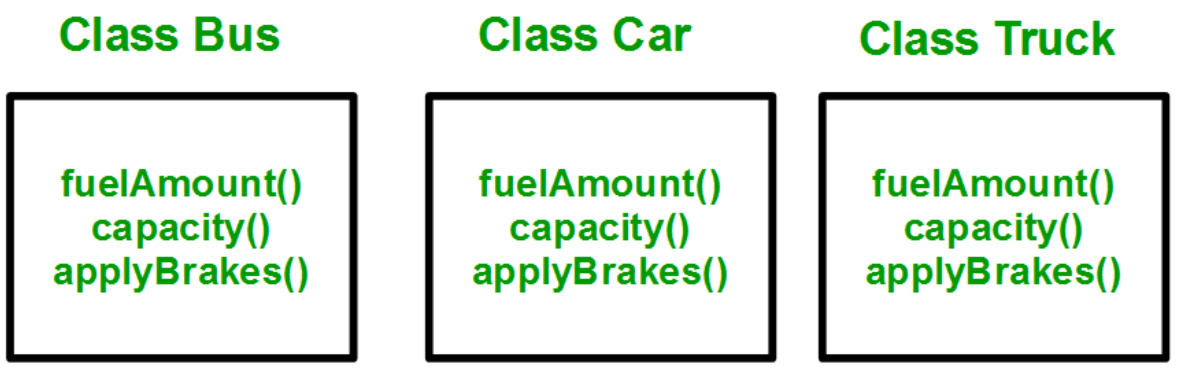
Inheritance is one of the most important features of Object-Oriented Programming. The capability of a class to derive properties and characteristics from another class is called Inheritance. The class whose properties are inherited is called base class or superclass whereas the class that inherits properties from another class is derived class or subclass.

Topics which we cover here are:

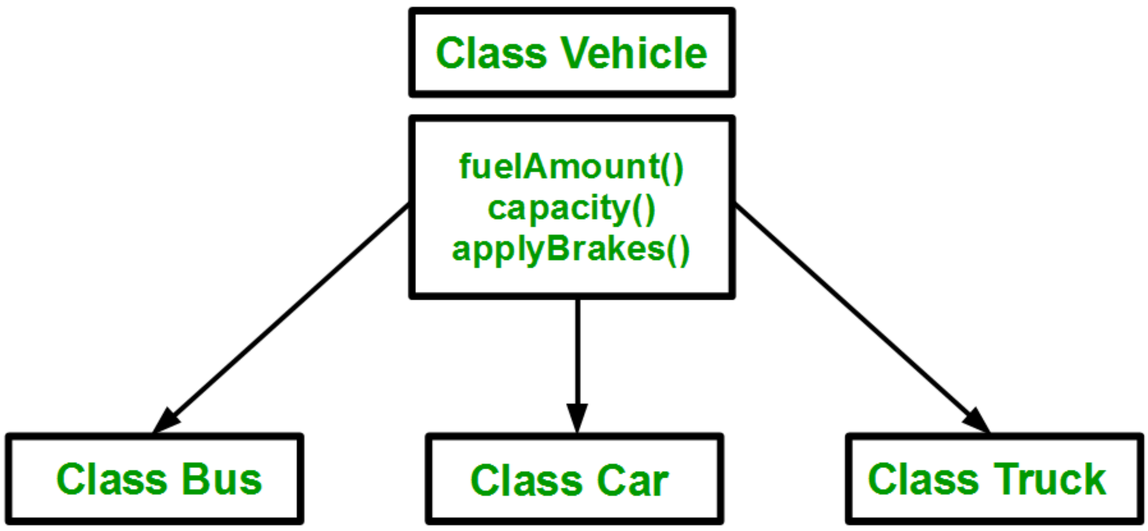
- Why and when to use inheritance?
- Modes of Inheritance
- Types of Inheritance

Why and when to use inheritance?

Let us consider a group of vehicles in which you have to create classes such as Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:



From the figure, it is clear that above process results in duplication of the same code 3 times This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability as shown in below figure:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

Syntax for implementing inheritance in C++:

For creating a sub-class that is inherited from the base class we have to follow the below syntax.

class derived_class_name : access_mode base_class_name

{

  // body of subclass

};

Here, subclass_name is the name of the subclass, access_mode is the mode in which you want to inherit the subclass for example public, private, etc. and base_class_name is the name of the base class from which you want to inherit the subclass.

Program 1:

Write a C++ program to implement inheritance.

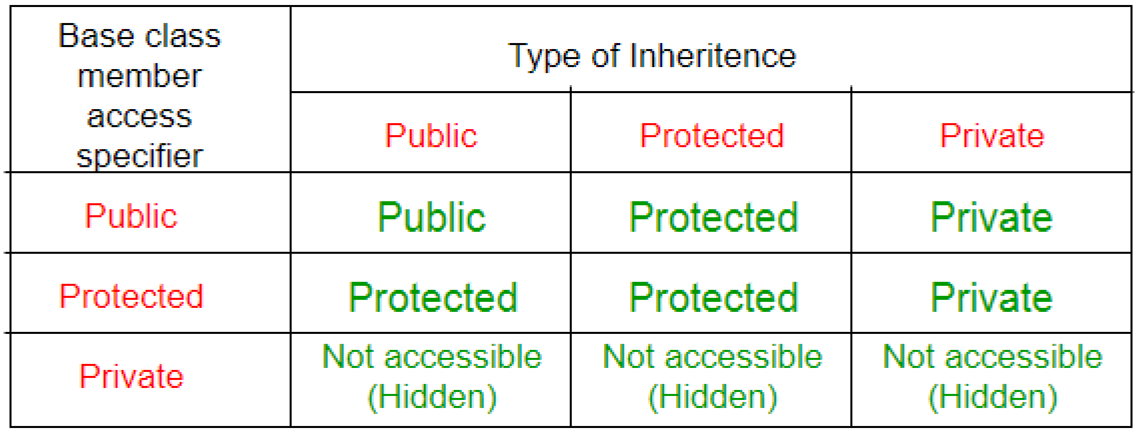
```
#include<iostream>
using namespace std;
class Parent {
public:
    int id_p;
};
class Child : public Parent {
public:
    int id_c;
};
int main()
{
    Child obj1;
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "Child id is: " << obj1.id_c << '\n';
    cout << "Parent id is: " << obj1.id_p << '\n';
    return 0;
}
```

Modes of Inheritance:

There are 3 modes of inheritance.

1. **Public Mode**: If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.
2. **Protected Mode**: If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
3. **Private Mode**: If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

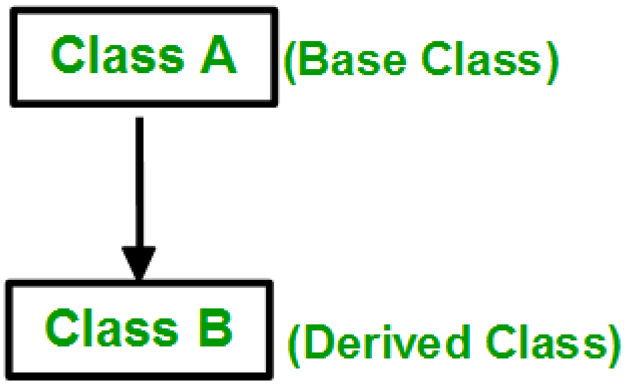
```
class A {
public:
    int x;

protected:
    int y;

private:
    int z;
};

class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```



Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.



Syntax:

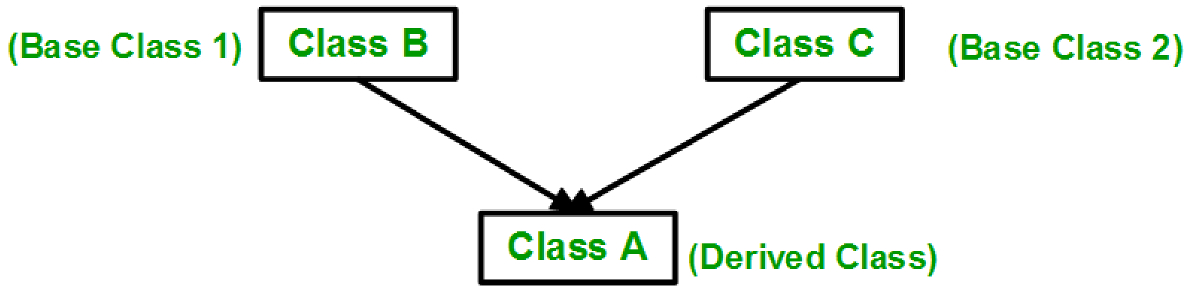
```
class subclass_name : access_mode base_class
{
  // body of subclass
};
```

Program 2:

Write C++ program to explain Single inheritance

```
#include<iostream>
using namespace std;
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle\n";
    }
};
class Car : public Vehicle {
};

// main function
int main()
{
    Car obj;
    return 0;
}
```

2. Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one subclass is inherited from more than one base class.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode
base_class2, ....
{
  // body of subclass
};
```

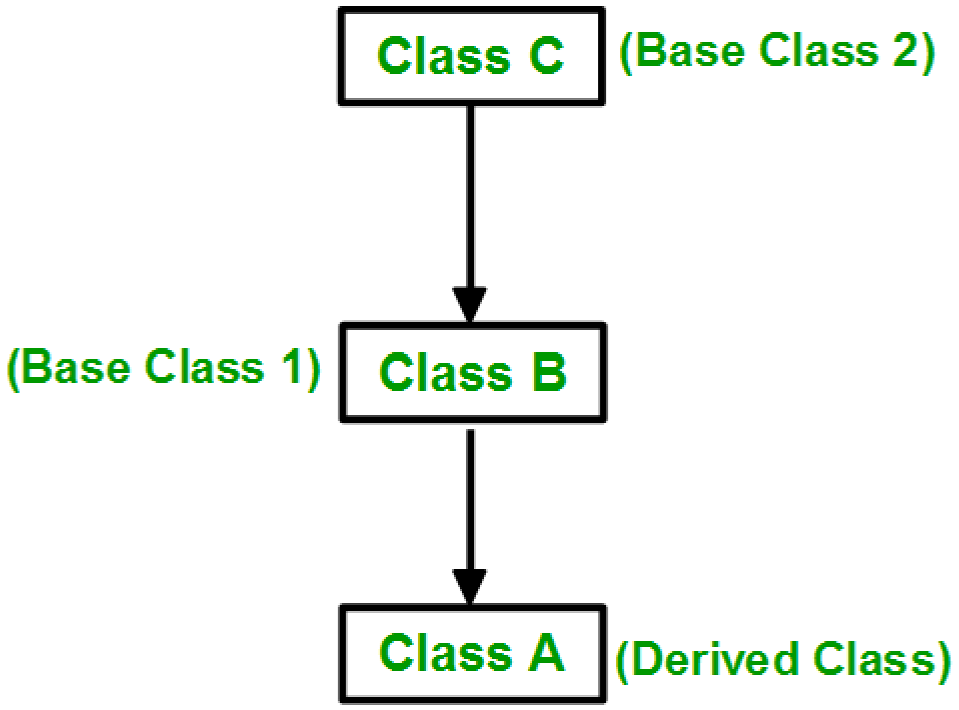
Program 3:

Write C++ program to explain Multiple inheritance

```
#include <iostream>
using namespace std;
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};
class FourWheeler {
```

```
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle\n";
    }
};
class Car : public Vehicle, public FourWheeler {
};
int main()
{
    Car obj;
    return 0;
}
```

3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.

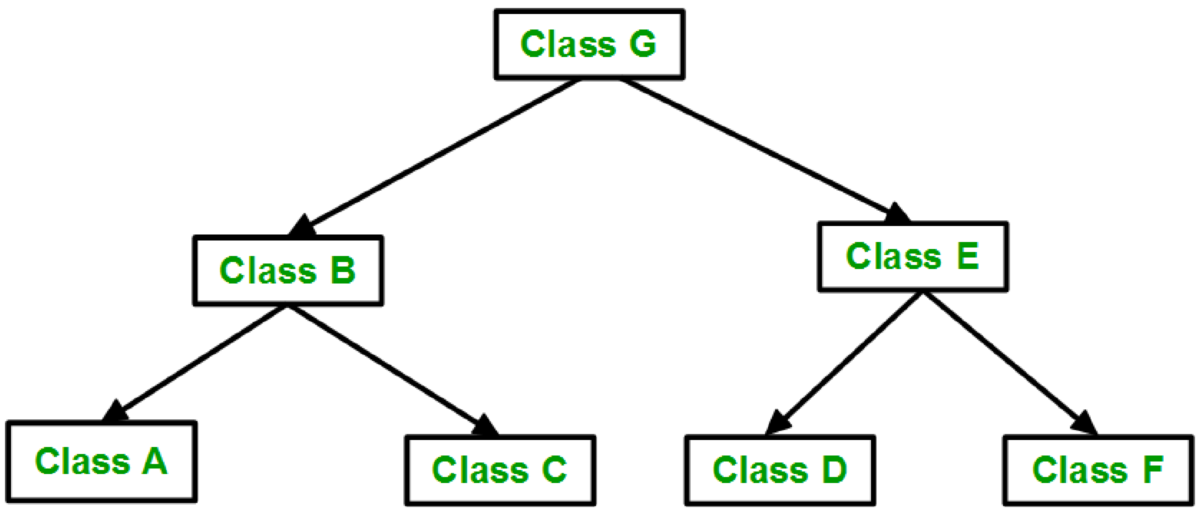


Program 4:

Write C++ program to explain Multilevel inheritance

```
#include <iostream>
using namespace std;
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};
class fourWheeler : public Vehicle {
public:
    fourWheeler()
    {
        cout << "Objects with 4 wheels are vehicles\n";
    }
};
class Car : public fourWheeler {
public:
    Car() { cout << "Car has 4 Wheels\n"; }
};
int main()
{
    Car obj;
    return 0;
}
```

4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.

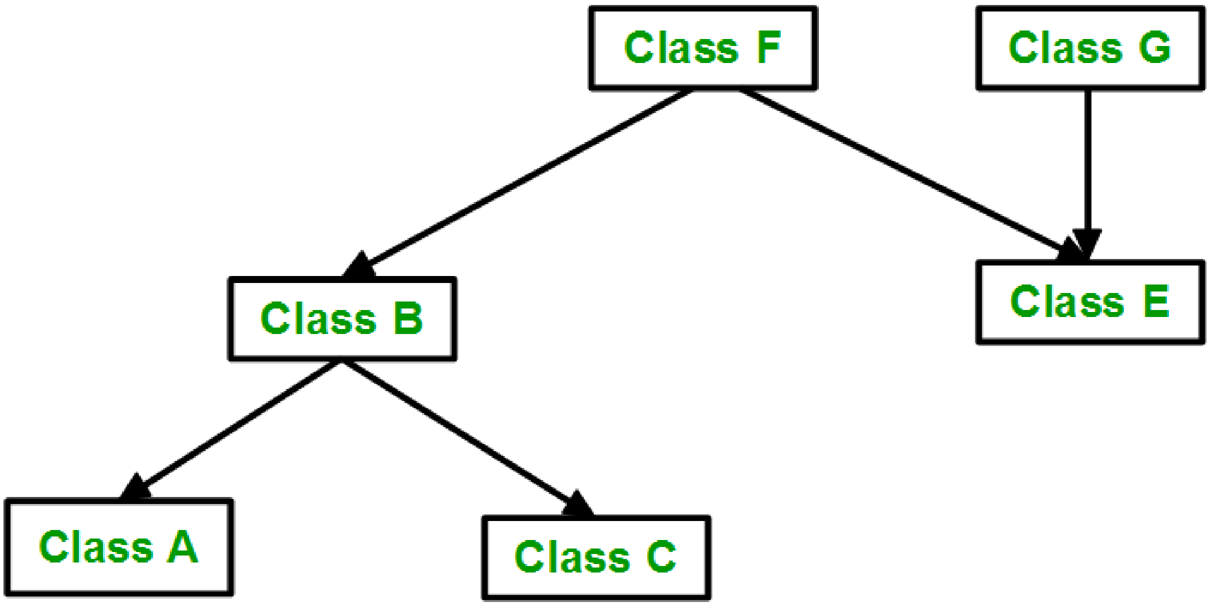


Program 5:

Write C++ program to explain Hierarchical inheritance

```
#include <iostream>
using namespace std;
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};
class Car : public Vehicle {
};
class Bus : public Vehicle {
};
int main()
{
    Car obj1;
    Bus obj2;
    return 0;
}
```

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritances:



Program 6:

Write C++ program to explain Hybrid inheritance

```
#include <iostream>
using namespace std;
// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// base class
class Fare {
public:
    Fare() { cout << "Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
};
```

```
// second sub class
class Bus : public Vehicle, public Fare {
};

// main function
int main()
{
    Bus obj2;
    return 0;
}
```

6. A special case of hybrid inheritance: Multipath inheritance:

A derived class with two base classes and these two base classes has one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.

Program 6:

Write C++ program to demonstrating ambiguity in Multipath Inheritance

```
#include <iostream>
using namespace std;

class ClassA {
public:
    int a;
};

class ClassB : public ClassA {
public:
    int b;
};

class ClassC : public ClassA {
public:
    int c;
};

class ClassD : public ClassB, public ClassC {
public:
    int d;
};

int main()
{
    ClassD obj;

    // obj.a = 10;                  // Statement 1, Error
    // obj.a = 100;                 // Statement 2, Error

    obj.ClassB::a = 10; // Statement 3
    obj.ClassC::a = 100; // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << " a from ClassB  : " << obj.ClassB::a;
    cout << "\n a from ClassC  : " << obj.ClassC::a;

    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

In the above example, both ClassB and ClassC inherit ClassA, they both have a single copy of ClassA. However Class-D inherits both ClassB and ClassC, therefore Class-D has two copies of ClassA, one from ClassB and another from ClassC.

If we need to access the data member of ClassA through the object of Class-D, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bcoz compiler can’t differentiate between two copies of ClassA in Class-D.

There are 2 Ways to Avoid this Ambiguity:

1) Avoiding ambiguity using the scope resolution operator: Using the scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statements 3 and 4, in the above example.

```
obj.ClassB::a = 10;       // Statement 3
obj.ClassC::a = 100;      // Statement 4
```

Note: Still, there are two copies of ClassA in Class-D.

2) Avoiding ambiguity using the virtual base class:

```
#include<iostream>

using namespace std;


class ClassA

{

  public:

    int a;

};


class ClassB : virtual public ClassA

{

  public:

    int b;

};


class ClassC : virtual public ClassA

{

  public:

    int c;

};


class ClassD : public ClassB, public ClassC
```

```
{
  public:
    int d;
};

int main()
{
    ClassD obj;

    obj.a = 10;       // Statement 3
    obj.a = 100;      // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n a : " << obj.a;
    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

According to the above example, Class-D has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given in statement 3.